REMARKS

Rejection of Claims on Prior Art Grounds in the 25 June 2004 Office Action and Traversal Thereof

In the 25 June 2004 Office Action, Claims 1-108 were rejected on prior art grounds, under 35 U.S.C. 102(e) and 35 U.S.C 103(a). Claims 1-3; 6-8, 11-14, 17–20, 23–26; 29–31; 34-37; 40–42; 45–47; 50–53; 56–59; 62–65; 68–70; 73–76; 79–82; 85–88; 91–94; 97–99; 102–105; and 108 are rejected under 35 U.S.C. 102(e) as being anticipated by Pazel, U.S. Patent 5,410,648 (Reissue 36,422). Claims 4, 5, 9, 10, 15, 16, 21, 22, 27, 28, 32, 33, 38, 39, 43, 44, 48, 49, 54, 55, 60, 61, 66, 67, 71, 72, 77, 78, 83, 84, 89, 90, 95, 96, 100, 101, 106, and 107 are rejected under 35 U.S.C. 103(a) as being unpatentable over Pazel, U.S. Patent 5,410,648 (Reissue 36, 422) in view of Graham, U.S. Patent 5,918,053. The above rejections on the stated art grounds are traversed, and consideration of the patentability of the claims 1-108, now amended, and new claims 109 – 128, is requested, in light of the following remarks.

Arguments for Patentability

The Examiner cites and strongly relies upon the Pazel patent to show previous disclosure of the present invention. Pazel describes a software *degugging* system or program that displays different views of a project source code. The Pazel program steps through the lines of code, highlights the "active" line of code, and opens a separate view with various view-types when another file is called by the code. These separate view-types comprise various textual representations (such as a source code listing or a

disassembly code listing) or optional graphical representations (such as a flow graph or a compressed image of the source code). Furthermore, the program allows the resurrection of discarded views without requiring record keeping since the program stores information about source code position, window characteristics, etc., which was an improvement over the prior art at the time.

Although this description of the Pazel patent may appear to reveal similarity to the present invention, the Pazel patent is readily distinguishable and teaches away from the present invention. The present invention describes a *software development tool* that displays simultaneously a graphical and a textual representation of the source code. Although the present invention does not exclude the use of a debugger program in association with the development tool, it nowhere relies on such use, and in fact, explicitly states that such functionality would be embodied by a separate tool (see page 31, lines 19-21 of the original patent application). Nowhere does the Pazel reference describe the use of its program to provide a software development platform like the present invention – Pazel merely and specifically describes a debugging system to identify causes of error and mistakes in existing source code files.

There are other substantial differences between the Pazel patent and the present invention. The present invention describes a software development tool that simultaneously displays matching textual and graphical representations of the source code. As distinguished from traditional methods of the prior art, the tool generates both views of the source code from a language-neutral representation of the source code stored in a transient meta-model. This arrangement allows for the immediate and automatic conversion of the alternate graphical or textual view of the code according to changes

made in the other view. This so-called "round-trip engineering" is a substantial improvement over the prior by overcoming the need for a repository as traditionally used in the prior art, which required a separate action to update the source code and had a number of disadvantages as outlined in the specification. This novel program structure and operation is nowhere described in the Pazel invention especially since these features are appropriate in the context of a software development tool and not a debugging program.

In addition to providing a software development tool that simultaneously creates textual and graphical representations of the source code, the present invention further provides a functionality that *demonstrates and tests the operation of the source code* by processing (i.e. executing, compiling, etc.) the software project line-by-line to structurally visualize the operation of the code and to identify causes of error in the execution of the program. This feature operates *within the context of the software development platform* by displaying the active line of code in a visually distinctive manner (such as highlighting) as it steps through the program.

It is true that the Pazel patent does describe a similar feature that highlights the active line of code while performing the debugger program. However, unlike the present invention, the Pazel program does not perform this feature in the context of a software development tool in the same textual and graphical views used for the manipulation of the software during the development process. Importantly, nowhere is it described by Pazel that the debugging program is capable of displaying a graphical representation of the project code such that *graphical elements are displayed in a visually distinctive manner* while the code is being processed. Furthermore, the only diagrammatic

representation of the source code vaguely disclosed in the Pazel patent is the "flow graph" that displays a *static* diagram of the program operation only after and as a product of the execution of the program by the debugger. This "flow graph" feature is not a part of a software development tool and is not capable of displaying its elements in a visually distinctive manner in association with its execution or processing.

The Examiner also relies on the Graham patent as further disclosing the present invention. The Graham patent was designed to overcome difficulties in communicating different aspects of a software development project between developers and outside parties and to overcome the common mismatch between software design and final implementation. The Graham invention accomplishes this task by creating a software development environment that includes a feature referred to as a "design virtual machine" built into a pane of the program window with the window also displaying a textual representation of the source code in a separate pane. When activated, the design virtual machine creates a *static* diagrammatic output of the software project in the graphical pane by stepping through the execution of the project code.

Again, this description of the Graham patent is substantially distinguishable from the present invention. First, although it has become common to have software development environments (i.e. UML) that display both diagrammatic and textual representations of the source code, the Graham patent does not describe the simultaneous and automatic updating of the graphical and textual views of the software project in the context of a software development tool as described in the present invention. Second, the Graham patent nowhere discloses the departure from traditional software development practice utilizing a repository to store working copies of the project code that must

separately be committed to update the source code file. Third, the Graham patent nowhere describes the presentation of either the "active" line of code in text or the corresponding graphical elements in a visually distinctive manner while processing the code.

The present invention provides a software development tool that displays both textual and diagrammatic representations of the source code utilizing a novel method that abolishes the need for a repository and that immediately and simultaneously updates the two views and the source code according to changes made to the software project. Additionally, the present invention provides a functionality that enables the programmer to structurally view the execution of the source code and identify the location of mistakes or errors within the same platform as the software development tool. The software development tool accomplishes this task on command by executing the code line-by-line while displaying either the "active" line of code and/or the corresponding graphical element in a visually distinctive manner such that the textual and graphical representations of the source code appear animated. Nowhere does the prior art combine these novel features into a single software development environment. Accordingly, the applicant asserts that the present invention provides an innovative, novel, and substantially improved method and system for software development over prior art.

CONCLUSION

For the foregoing reasons, claims 1-128, now amended, constituting the claims pending in the application, are submitted to be fully patentable and in allowable condition

to address and overcome the rejections. If any issues remain outstanding, incident to the allowance of the application, Examiner Shrader is respectfully requested to contact the undersigned attorney at (919) 664-8222 or via email at jinang@trianglepatents.com to . discuss the resolution of such issues, in order that prosecution of the application may be concluded favorably to the applicant, consistent with the applicant's making of a substantial advance in the art and particularly pointing out and distinctly claiming the subject matter that the applicant regards as the invention.

This Office Action response is submitted to the USPTO via USPS Express Mail on September 24, 2004.

Respectfully submitted,

JiNan Glasgow #42585
Glasgow Law Firm, PLLC
PO Box 28539
Raleigh, NC 27611-8539
919-664-8222
919-664-8625 (fax)

22 pg

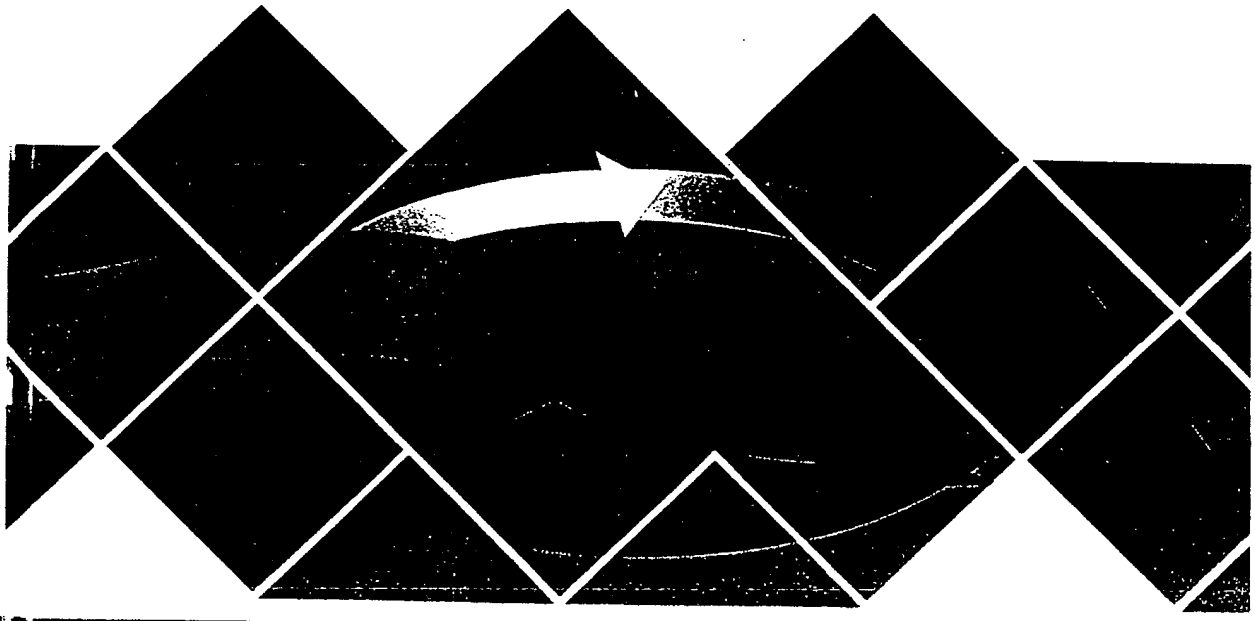
Ivar Jacobson
Magnus Christerson Patrik Jonsson
Gunnar Övergaard

COMPUTER LANGUAGE Productivity Award Winner

January 19, 1996

Object-Oriented Software Engineering

A Use Case Driven Approach



acm PRESS
ADDISON-WESLEY

1/9/1
DIALOG(R)File 4
(c) 1998 R.R.Bo

04610846 0103
TITLE: Object-
AUTHOR: Jacob
ISBN: 0-201-5
STATUS: Act
PUBLISHER:
PUBLICATION
BINDING: Cl
(Retail Pri
ISBN: 0-201-4
STATUS: Act
EDITION: 2n
PUBLISHER:
PUBLICATION
BINDING: Cl
PAPERBOUND
TECHNOLOGY
DATE IN FILE:
INTELLECTUAL
SUBFILE: ST (
LIBRARY OF CO
OBJECT-ORIE

Reprin

Date

1/9/1
DIALOG(R)File 470:Books in Print(R)

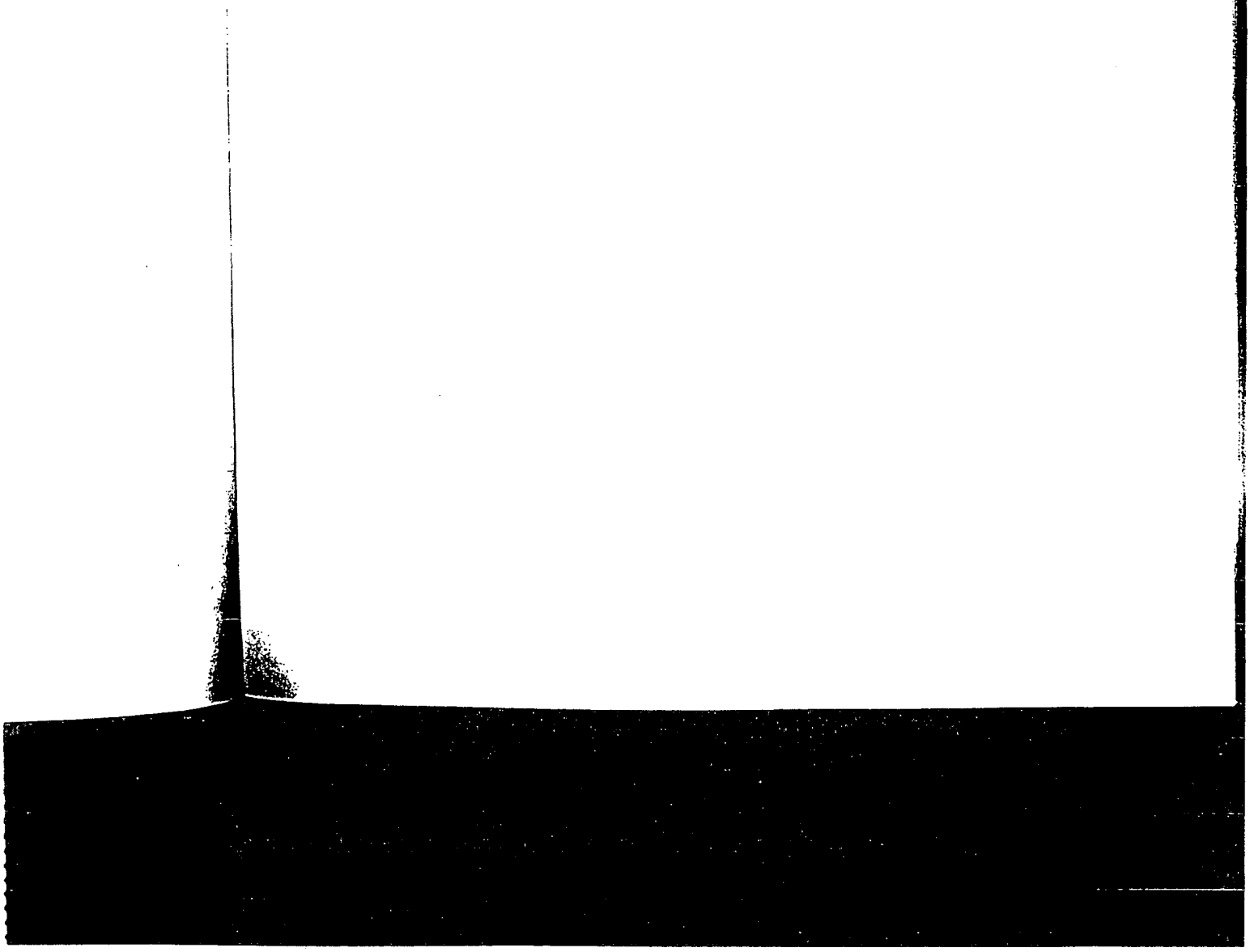
04610846 01039986
TITLE: Object-Oriented Software Engineering
AUTHOR: Jacobson, Ivar
ISBN: 0-201-54435-0 (1992/04)
STATUS: Active Record
PUBLISHER: Addison-Wesley
PUBLICATION DATE: 011992 (199201)
BINDING: Cloth Text - $48.95 (Ingram Price), $30.84 (Net), $48.95 (Retail Price)
ISBN: **0-201-40347-1** (1995/10)
STATUS: Active Record
EDITION: 2nd ed.
PUBLISHER: Addison-Wesley
PUBLICATION DATE: 011996 (199601)
BINDING: Cloth Text - (Write/Call Publisher for information)
PAPERBOUND BOOK SUBJECT HEADINGS: TECHNOLOGY-COMPUTERS AND COMPUTER TECHNOLOGY (00456X)
DATE IN FILE: 1992/04
INTELLECTUAL LEVEL: College Audience
SUBFILE: ST (Science & Technical)
LIBRARY OF CONGRESS SUBJECT HEADINGS: SOFTWARE ENGINEERING (42029454); OBJECT-ORIENTED PROGRAMMING (COMPUTER SCIENCE) (42041697)

Reprint → Janvary 19, 1996

Date Published Janvary, 19, 1992

Object-Oriented Software Engineering

A Use Case Driven Approach



ACM PRESS

Editor-in-Chief	**Peter Wegner**	Brown University
International Editor (Europe)	**Dines Bjørner**	Technical University of Denmark

SELECTED TITLES

Object-Oriented Reuse, Concurrency and Distribution *Colin Atkinson*

Advances in Database Programming Languages *François Bancilhon and Peter Buneman (Eds)*

Software Reusability (Volume 1: Concepts and Models) *Ted Biggerstaff and Alan Perlis (Eds)*

Software Reusability (Volume 2: Applications and Experience) *Ted Biggerstaff and Alan Perlis (Eds)*

Object-Oriented Concepts, Databases and Applications *Won Kim and Frederick H. Lochovsky* (Eds)

Distributed Systems *Sape Mullender (Ed)*

The Oberon System: User Guide and Programmer's Manual *Martin Reiser*

Programming in Oberon: Steps Beyond Pascal and Modula *Martin Reiser and Niklaus Wirth*

The Programmer's Apprentice *Charles Rich and Richard C. Waters*

Instrumentation for Future Parallel Computer Systems *Margaret Simmons, Ingrid Bucher and Rebecca Koskela (Eds)*

User Interface Design *Harold Thimbleby*

Advanced Animation and Rendering Techniques: Theory and Practice *Alan Watt and Mark Watt*

Project Oberon: The Design of an Operating System and Compiler *Niklaus Wirth and Jurg Gutknecht*

Ob

E

Har
New
Toky

Object-Oriented Software Engineering

A Use Case Driven Approach

IvarJacobson
Magnus Christerson Patrik Jonsson
Gunnar Övergaard

Objective Systems SF AB
PO Box 1128, S-16422 Kista, Sweden

Objectory Corp.
PO Box 2630, Greenwich, CT 06836, USA

ADDISON-WESLEY

Harlow, England • Reading, Massachusetts • Menlo Park, California
New York • Don Mills, Ontario • Amsterdam • Bonn • Sydney • Singapore
Tokyo • Madrid • San Juan • Milan • Mexico City • Seoul • Taipei

Addison Wesley Longman Limited
Edinburgh Gate
Harlow
Essex, CM20 2JE
England

Cover designed by Chris Eley
Typeset by Columns Design & Production Services Ltd, Reading
Printed in the United States of America
ISBN 0-201-54435-0

First printed 1992. Reprinted 1992 and 1993. Revised fourth printing 1993.
Reprinted 1994 (twice), 1995 (twice), 1996, 1997, and 1998.

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library.

Library of Congress Cataloging-in-Publication Data available.

Foreword

Ivar Jacobson is in my opinion one of the foremost methodologists in the field of software engineering. I take great pleasure in writing this, because he is also a close personal friend. He brings a refreshingly pragmatic point of view to a discipline that often seems be so abstract and arcane as to be hopelessly remote from the real world of 'blue collar' programmers. His methodology is based on some really innovative ideas about modeling the software process, presented within a tried and proven engineering framework. It brings to the task of analyzing, designing and constructing complex software intensive products the same disciplined approach that is to be found in other branches of engineering.

Along with many others I have urged Ivar for some time to publish his methodology in a textbook, so that it would be accessible to a larger audience. I believe that the concepts in Objectory, the first comprehensive object-oriented process for developing large scale industrial systems, are important and should get wider exposure. This book represents over 20 years of experience building real software based products and a great deal of serious thinking about how such systems should be built. If you have any interest at all in software you will enjoy reading it.

Objectory stands out as being a truly object-oriented methodology, in which both the process and the methodology are themselves represented as objects. While some may find this idea of a reflective or 'meta' architecture to be rather exotic, it is in fact intensely practical and absolutely essential. It makes Objectory an extensible methodology which can be specialized to both the organization and the application domains. Simply put, Objectory provides a software process for building not just software, but also other more specialized software processes.

Another key innovation in Objectory is the concept of use cases, which has now been proved effective in a number of real-world projects. Use cases provide the needed linkage between requirements,

development, testing and final customer acceptance. This idea, which originated in Ivar's work on the AXE switch, has been generalized so that it can be applied in application domains as diverse as command and control and business information systems.

Use cases provide a concrete representation of software requirements, which allows them to be both formally expressed and systematically tested. Changes in requirements map directly onto changes in the set of use cases. In this way Objectory provides a solid methodological foundation for rapid prototyping and other forms of incremental software development. Objectory enables managers to move beyond labour intensive hand assembly of software systems, and allows them to transform their organizations into highly automated factories to manufacture software from reusable components.

Many feel that we are in the midst of a software crisis, and I agree. High-quality software has become one of the most sought after commodities in the modern world. We just can't seem to get enough of it, on time and on budget, to meet the demand. This book will help you overcome the software crisis in your own organization, by showing you how to make software construction into a reliable and predictable engineering activity.

One of the more profound insights offered by modern software engineering is that change is inevitable, and that software must be designed to be flexible and adaptable in the face of changing requirements. Objectory, with its reflective architecture, goes one step further, and provides an extensible methodology which can itself adapt to shifts in the business climate or the demands of new technologies. No static text can ever capture all the nuances of such a dynamic software entity but this one comes very close. I strongly recommend it, not only for software managers and designers, but for anyone who wishes to understand how the next generation of software systems should be built.

Dave Thomas

Foreword

Ivar Jacobson has taken the time to create a book that is certain to become essential reading for software developers and their managers. In this book, Jacobson establishes a new direction for the future of software engineering practice. It is a thoughtful and thorough presentation of ideas and techniques that are both solidly proven and, simultaneously, at the leading edge of software engineering methodology. Jacobson is simply a thinker who has been ahead of his time in creating usable methods for building better, more reliable and more reusable large software systems.

Despite the title, this is not 'another book on object-oriented analysis and design', nor yet another standard reworking on the word-of-the-week. Once, of course, the word-of-the-week in software engineering was 'modular', later it was 'structured', and now, as every programmer or software engineer who reads or attends conferences knows, it is 'object-oriented.'

When the word-of-the-week was still 'structured', and I wrote the first edition of *Structured Design*, the very idea of systematic methods for software development was radical. Software engineering was in its infancy, and when I introduced data flow diagrams and structure charts, few recognized either the need for notation or the benefits of well-conceived modeling tools for analysis and design.

But things have changed. Now, new methodologies are created over cocktails, and books spin out of word-processors so fast that revised or 'corrected' editions appear almost before the original has reached the bookstores. Since nearly everyone now recognizes that a methodology must be supported by a notation, notations proliferate. A new object-oriented design notation can be churned out over a weekend so long as the major objective is simply squiggles and icons with a unique 'look and feel', and issues of usability and power in modeling are considered unimportant.

And here we have yet another notation supporting one more methodology? Not quite.

It is true that the serious reader will have to surmount both new

terminology and new notation to get to the marrow, but this book is different. It was not conceived and written overnight. The methodology it describes has been in use for years to design and build numerous software systems, and its notation has evolved slowly from both manual and CASE-supported application. It is not the work of a writer or consultant with a long booklist, but comes from a practising software engineer and leader in software engineering who has been doing large-scale object-oriented development for longer than most people even knew that objects existed. Throughout this period, the ideas and methods have been honed by the grindstone of building software and refined by thoughtful reflection and analysis.

What we have here is an approach to object-oriented analysis and design that is fundamentally different from most of the highly touted and more visible methods that clutter the landscape. I believe it is an approach of proven power and even greater promise.

The real power of this approach rests not only in the wealth of experience on which it is based but also in the way in which it starts from a different point of departure and builds an entirely different perspective on how to organize software into objects. Jacobson does not build naive object models derived from simplistic reinterpretations of data modeling and entity object relationship models. He starts from an entirely different premise and set of assumptions uniquely tailored to creating robust, sophisticated object structures that stand the test of time.

His approach centers on an analysis of the ways in which a system is actually used, on the sequences of interactions that comprise the operational realities of the software being engineered. Although it fully incorporates the conceptual constructs, the application and enterprise entities that undergird our thinking about software systems, it does not force the entire design into this rigid pattern. The result is a more robust model of an application, leading to software that is fundamentally more pliant, more accommodating to extensions and alterations and to collections of component parts that are, by design, more reusable.

At the heart of this method is a brilliantly simple notion: the use case. A use case, as the reader will learn, is a particular form or pattern or exemplar of usage, a scenario that begins with some user of the system initiating some transaction or sequence of interrelated events. By organizing the analysis and design models around user interaction and actual usage scenarios, the methodology produces systems that are intrinsically more useable and more adaptable to changing usage. Equally important, this approach analyzes each use case into its constituent parts and allocates these systematically to

software objects in such a way that external behavior and internal structure and dynamics are kept apart, such that each may be altered or extended independently of the other. This approach recognizes not one kind of object, but three, which separate interface behavior from underlying entity objects and keeps these independent of the control and coordination of usage scenarios.

Using this approach, it is possible to construct very large and complex designs through a series of small and largely independent analyses of distinct use cases. The overall structure of the problem and its solution emerges, step-by-step and piece-by-piece, from this localized analysis. In principle – and in practice – this methodology is one whose power increases rather than diminishes with the size of the system being developed.

Use case driven analysis and design is a genuine breakthrough, but it is also well-grounded in established fundamentals and connected to proven ideas and traditions in software engineering in general and object-oriented development in particular. It echoes and extends the popular model–view–controller paradigm of object-oriented programming. It is clearly kin to the event-driven analysis and design approaches of Page-Jones and Weiss, as well as to the widely practised event-partitioning methods pioneered by McMenamin and Palmer.

On this ground, Ivar Jacobson has built a work that is nothing short of revolutionary. Rich with specific guidelines and accessible examples, with completely detailed case studies based on real-world projects, this book will give developers of object-oriented software material that they can put into practice immediately. It will also challenge the reader and, I am confident, enrich the practise of our profession for years to come.

Larry L. Constantine

Preface

This is a book on industrial system development using object-oriented techniques. It is not a book on object-oriented programming. We are convinced that the big benefits of object orientation can be gained only by the consistent use of object orientation throughout all steps in the development process. Therefore the emphasis is placed on the other parts of development such as analysis, design and testing.

You will benefit from this book if you are a system developer seeking ways to improve in your profession. If you are a student with no previous experience in development methods, you will learn a robust framework which you can fill with details as you take part in future development projects. Since the focus of the text is on development, the book will be convenient to use in combination with other texts on object-oriented programming. Many examples illustrate the practical application of analysis and design techniques.

From this book you will get a thorough understanding of how to use object orientation as the basic technique throughout the development process. You will learn the benefits of seamless integration between the different development steps and how the basic object-oriented characteristics of class, inheritance and encapsulation are used in analysis, construction and testing. With this knowledge you are in a much better position to evaluate and select the way to develop your next data processing system.

Even though object orientation is the main theme of this book, it is not a panacea for successful system development. The change from craftsmanship to industrialization does not come with the change to a new technique. The change must come on a more fundamental level which also includes the organization of the complete development process. Objectory is one example of how this can be done.

This book does *not* present Objectory. What we present is the fundamental ideas of Objectory and a simplified version of it. In this book we call this simplified method OOSE to distinguish it from Objectory. To use the process in production you will need the complete

and detailed process description which, excluding large examples, amounts to more than 1200 pages. Introducing the process into an organization needs careful planning and dedication. It also requires that the process be adapted to the unique needs of the organization. Such process adaptations must of course be carefully specified, which can be done in a development case description, as will later be explained.

It is our hope that we have reached our goal with this book, namely to present a coherent picture of how to use object-orientation in system development in a way which will make it accessible both to practitioners in the field and to students with no previous knowledge of system development. This has been done within a framework where system development is treated as an industrial activity and consequently must obey the same requirements as industry in general. The intention is to encourage more widespread use of object-oriented techniques and to inspire more work on improving the ideas expounded here. We are convinced that using these techniques will lead to better systems and a more industrial approach to system development.

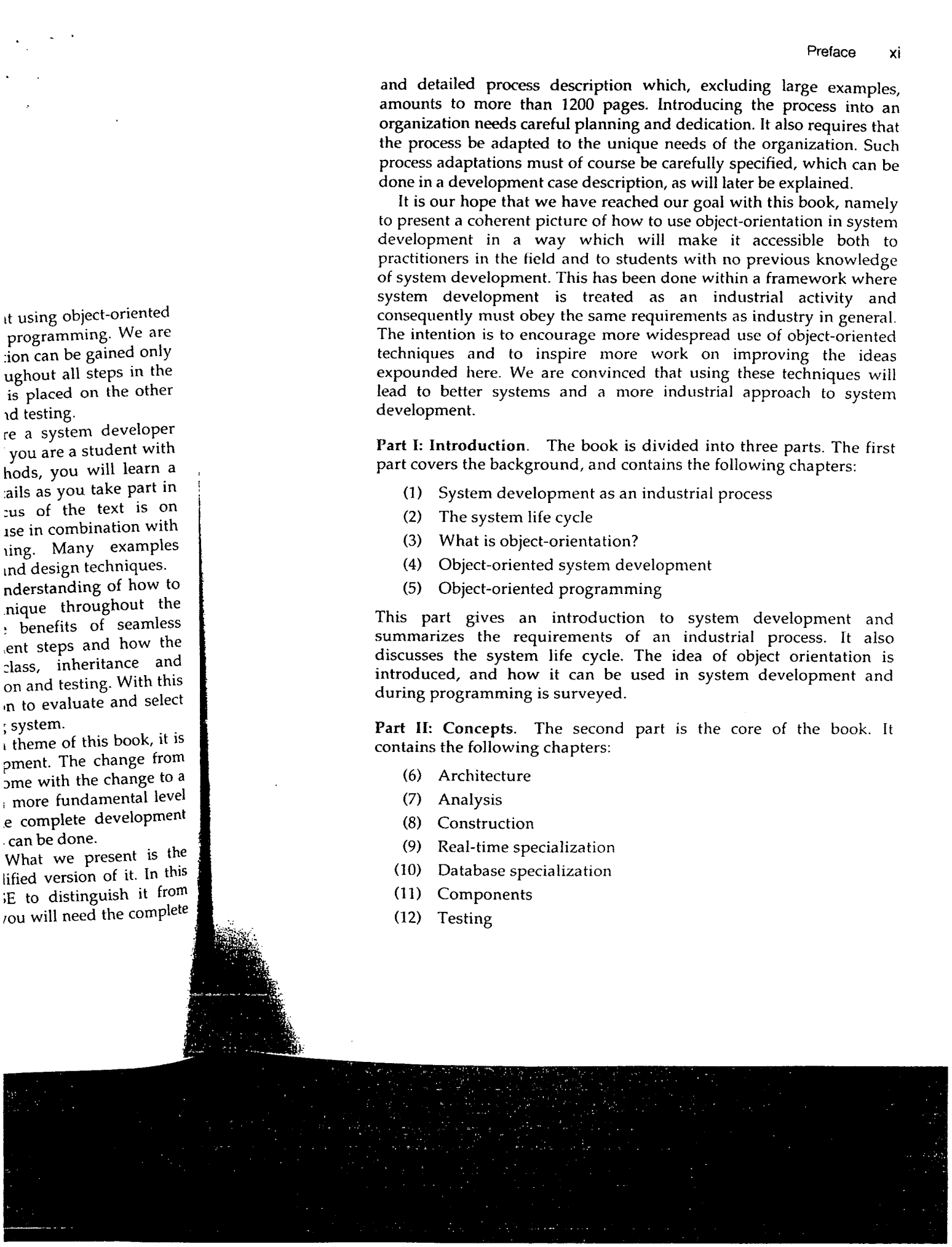
Part I: Introduction. The book is divided into three parts. The first part covers the background, and contains the following chapters:

(1) System development as an industrial process
(2) The system life cycle
(3) What is object-orientation?
(4) Object-oriented system development
(5) Object-oriented programming

This part gives an introduction to system development and summarizes the requirements of an industrial process. It also discusses the system life cycle. The idea of object orientation is introduced, and how it can be used in system development and during programming is surveyed.

Part II: Concepts. The second part is the core of the book. It contains the following chapters:

(6) Architecture
(7) Analysis
(8) Construction
(9) Real-time specialization
(10) Database specialization
(11) Components
(12) Testing



The first chapter in this part introduces the fundamental concepts of OOSE and explains the reason why these concepts are chosen. The following chapter discuss the method of analysis and construction. The next two chapters discuss how the method may be adapted to real-time systems and database management systems. The components chapter discusses what components are and how they can be used in the development process. Testing activities are discussed in a chapter of their own.

Part III: Applications. The third and last part covers applications of OOSE and how the introduction of a new development process may be organized and managed. This part ends with an overview of other object-oriented methods. This part comprises:

(13) Case study: warehouse management system
(14) Case study: Telecom
(15) Managing object-oriented software engineering
(16) Other object-oriented methods

Appendix. Finally we have an appendix which comments on our development of Objectory.

So, how should you read this book? Of course, to get a complete overview, the whole book should be read, including the appendix. But if you want to read only selected chapters the reading cases below could be used.

If you are an experienced object-oriented software engineer, you should be familiar with the basics. You could read the book as suggested in Figure P.1.

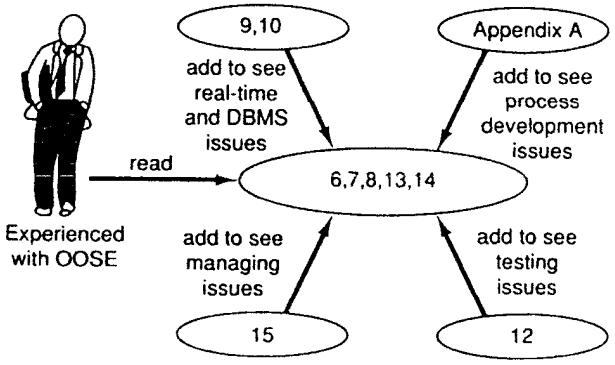


Figure P.1

amental concepts of
pts are chosen. The
is and construction.
may be adapted to
ent systems. The
s are and how they
sting activities are

overs applications of
opment process may
an overview of other

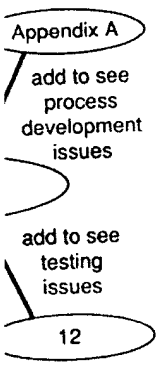
tem

neering

ch comments on our

se, to get a complete
luding the appendix.
rs the reading cases

ftware engineer, you
ld read the book as



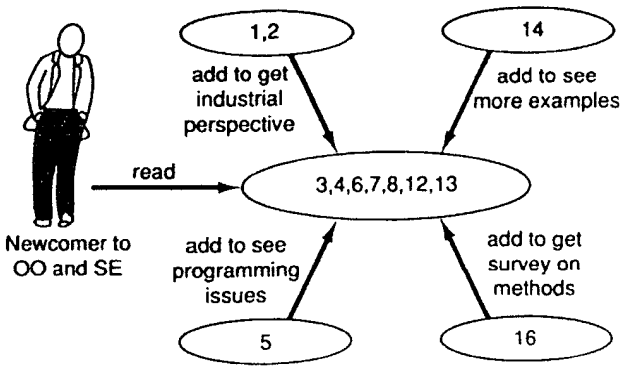


Figure P.2

If you are a newcomer to object-orientation and software engineering you could read the book as in Figure P.2.

If you are an experienced software engineer you could read the book as in Figure P.3.

If you are a manager you could read the book as proposed in Figure P.4.

Although the book is not object-oriented, it is written in a modularized way and can be configured in several different ways. Building systems in this way is the theme of the book, and the technique and notation used above is very similar to the technique used in this book.

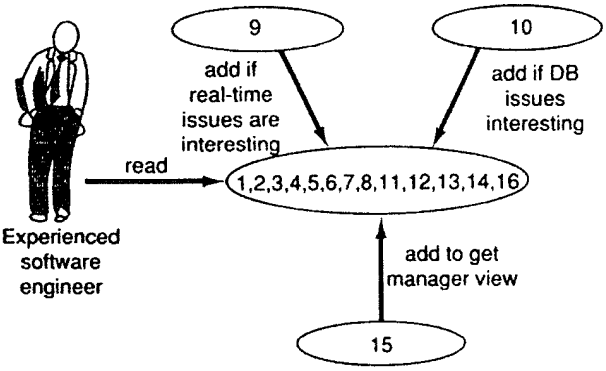


Figure P.3

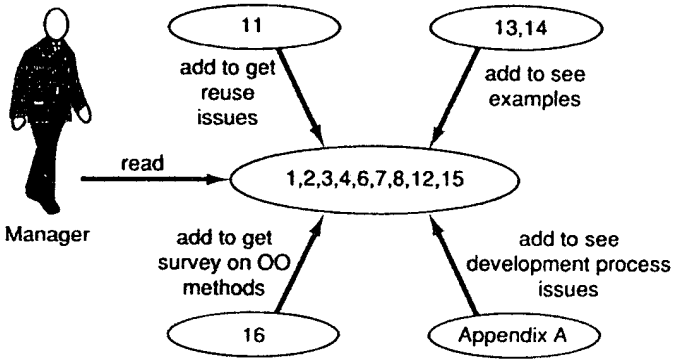


Figure P.4

A short history and acknowledgements

The work presented in this book was initiated in 1967 when I proposed a set of new modeling concepts (notation with associated semantics) for the development of large telecommunication switching systems. The main concepts were signals and blocks. A real-time system is an open system communicating with its environment by signals alone. A signal models the physical stimulus/response communication which a concrete system has when interacting with the outside world. Given a signal as input, a system performs internal actions such as executing algorithms, accessing internal information, storing results and sending output signals to the environment. This view presents the system in a very abstract way – as a black box. A less abstract view on a lower level models the system as a set of interconnected blocks. Blocks are modules which can be implemented in hardware or software or any combination of both. A block communicates with its environment only through signals. Signals between two blocks are internal, whereas signals modeling physical communication, that is, signals between a block and the environment of the system, are external. Internal signals are messengers conveying data from one block to another within the same system. All entries of a block were labelled and constituted the signal interface of that block, to be specified in a separate interface document. Hence the system can now be viewed as a set of interconnected blocks jointly offering the functions of the system. Each block has a program which it obeys on receipt of an input signal, performing internal actions, that is, executing algorithms, storing and accessing block internal information, and sending internal and external signals to the environment.

The proposal can be summarized as an attempt to unify long experience of systems design with the possibilities offered by dramatically new computer technology. Since the two technologies were so different, this was not a self-evident method, neither within Ericsson nor within computer science. There was a rather strong belief that the two represented unrelated technological universes: the new one was so different that it would be meaningless and only a burden to make any attempt to learn from the old one. However, the two techniques were joined and a set of modeling concepts evolved.

The modeling constructs were soon followed by the skeleton of a new design method, the use of which was first demonstrated in the development of the AKE system put into service in Rotterdam in 1971, and more completely demonstrated in the AKE system put into service in Fredhall, Sweden, in 1974. Naturally this experience has guided subsequent work on the development of the successor to AKE, the AXE system, which is now in use in more than 80 countries worldwide. The modeling constructs were very important and, for the AXE system, a new programming language and a new computer system were developed in accordance with these early ideas.

Although it is a neighbouring country, the early development of object-oriented programming and Simula in the 1960s in Norway was done independently and in parallel with our work. It was not until 1979 that we 'discovered' object-oriented programming and then it was in terms of Smalltalk. Although object-oriented ideas have influenced our recent work, basically two separate problems are being solved: 'large-scale' and 'small-scale'.

The modeling constructs introduced during the 1960s were further formalized in research taking place between 1978 and 1985. This research resulted in a formally described language which offered support for object-orientation with two types of object and two types of communication mechanism, send/wait and send/no-wait semantics. The language supported concurrency with atomic transactions and a special semantic construction for the handling of events similar to the use case construct presented later. This work, reported in a PhD thesis in 1985, resulted in a number of new language constructs, initially developed from experience, being refined and formalized. This was a sound basis from which to continue and, taking a new approach, develop the method. The principles of Objectory were developed in 1985–7. I then further refined and simplified the ideas, generalized the technique used in the telecom applications, extended it with the inheritance concept and other important constructs like extensions, and coupled to it an analysis technique and object-oriented programming.

Today these concepts have been further refined. The Objectory process, of which this book describes some fundamental ideas, is the result of work by many individuals, most of whom today work at Objective Systems SF AB, Sweden. Gunnar Övergaard and Patrik Jonsson did much of the writing of the first process description of Objectory analysis and design, respectively. Magnus Christerson did much to condense and rewrite the material into the form of this book. They have all contributed to Objectory; especially the formalization of the concepts. Magnus has also related the ideas of Objectory to other areas as presented in this book. Fredrik Lindström has also been involved in the condensation of the material for this book. Agneta Jacobson, Bud Lawson and Lars Wiktorin have prepared material for some of the chapters.

Marten Gustafsson has substantially contributed to the analysis part of Objectory. Valuable contributions to Objectory have also been made by the following people: Sten-Erik Bergner, Per Björk, Ann Carlbrand, Håkan Dyrhage, Christian Ehrenborg, Agneta Jacobson, Sten Jacobson, Mikael Larsson, Fredrik Lindstrom, Lars Lindroos, Benny Odenteg, Karin Palmkvist, Janne Pettersson, Birgitta Spiridon, Per Sundquist, Lars Wetterborg and Lars Wiktorin. The following users of Objectory have also contributed by feeding back experiences and ideas to enable improvements: Staffan Ehnebom, Per Hedfors, Jörgen Hellberg, Per Kilgren, Håkan Lidström, Christian Meck, Christer Nilsson, Rune Nilsson, Göran Schefte, Fredrik Strömberg, Karin Villers, Stefan Wallin and Charlotte Wranne. The following persons have done a lot to support the technology described in this book: Kjell S. Andersson, Hans Brandtberg, Ingemar Carlsson, Håkan Dahl, Gunnar M. Eriksson, Björn Gullbrand, Lars Hallmarken, Bo Hedfors, Barbara Hedlund, Håkan Jansson, Christer Johansson, Ingemar Johnsson, Kurt Katzeff, Rolf Leidhammar, Jorma Mobrin, Jan-Erik Nordin, Anders Rockström, Kjell Sörme, Göran Sundelöf, Per-Olof Thysselius, Ctirad Vrana and Erik Örnulf. The following people have given me strong personal inspiration and support: Dines Bjørner, Tore Bingefors, Dave Bulman, Larry Constantine, Göran Hemdal, Tom Love, Nils Lennmarker, Lars-Olof Norén, Dave Thomas and Lars-Erik Thorelli. In Sweden we do not normally thank family and friends in these circumstances, but no one believes that results like these can be achieved without exceptional support from them. We are also grateful to the support we have been given from STU (Swedish National Board on Industrial Development, now reorganized to NUTEK) through the IT-4 program which has been part of the financial support and sponsorship for the writing of this book.

Changes to this revised printing, apart from minor general corrections and improvements, are:

- The testing chapter has been restructured and in parts rewritten, also an emphasis on early testing has been added.
- The discussion of robust object structures has been increased and also an example has been added. We hope this will better clarify why such an object structure gives more robust systems.
- The notion of a development case has been introduced as a way to adapt a general process to the specific needs of an organization or a project.
- Some people we would like to thank were unfortunately left out in the first printing and have now been added to the acknowledgement section, particularly Dave Bulman and Nils Lennmarker who have inspired the technology presented in this book.

The authors

efined. The Objectory
ıdamental ideas, is the
whom today work at
Övergaard and Patrik
process description of
lagnus Christerson did
o the form of this book.
ılly the formalization of
ıs of Objectory to other
ıdström has also been
l for this book. Agneta
e prepared material for

uted to the analysis part
ory have also been made
r Björk, Ann Carlbrand,
Jacobson, Sten Jacobson,
ıdroos, Benny Odenteg,
piridon, Per Sundquist,
wing users of Objectory
nces and ideas to enable
rs, Jörgen Hellberg, Per
Christer Nilsson, Rune
ırin Villers, Stefan Wallin
ons have done a lot to
ook: Kjell S. Andersson,
ahl, Gunnar M. Eriksson,
dfors, Barbara Hedlund,
r Johnsson, Kurt Katzeff,
ordin, Anders Rockström,
selius, Ctirad Vrana and
iven me strong personal
Bingefors, Dave Bulman,
e, Nils Lennmarker, Lars-
relli. In Sweden we do not
circumstances, but no one
eved without exceptional
the support we have been
n Industrial Development,
-4 program which has been
hip for the writing of this

art from minor general

Contents